Funktionale Programmierung

Typklassen

Andreas Abel und Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

17. April 2012

Prinzipale Typen überladener Operatoren

- Die Operation + ist in SML **überladen** (engl. *overladed*).
- Sie auf `int` und `real` angewendet werden.

```
+ : int -> int -> int
+ : real -> real -> real
```

- Im mehrdeutingen Fall `fn x => fn y => x + y` wählt SML willkürlich `int -> int -> int`.
- Der **prinzipale Typ** von + wäre jedoch (in hypothetischer Syntax):

$$\texttt{+ : (}\alpha \in \{\texttt{int,real}\}\texttt{) => } \alpha \texttt{ -> } \alpha \texttt{ -> } \alpha$$

- Definierten wir das Prädikat "numerischer Typ" $\texttt{Num } \alpha$ als $\alpha \in \{\text{int,real}\}$, so könnten wir schreiben:

$$\texttt{+ : (Num } \alpha\texttt{) => } \alpha \texttt{ -> } \alpha \texttt{ -> } \alpha$$

Überladene Operatoren in Haskell

- Der Haskell-Interpreter GHCI liefert auf die Typabfrage `:type (+)`

```
(+) :: Num a => a -> a -> a
```

- Dies besagt, dass `(+)` den Typen `a -> a -> a` hat falls `a` der **Typklasse `Num`** angehört.
- Die Typklasse **`Num`** ist folgendermassen definiert (einsehbar mit `:info Num`):

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  ...
```

- Auch Zahlkonstanten sind überladen, siehe z.B. **`:type`** `5`.

```
5 :: Num a => a
```

Überladene Gleichheit in Haskell

- Test auf Gleichheit ist das überladene Symbol ==.
- Keine Sonderbehandlung wie in SML (Gleichheitstypen) nötig.
- Gleichheit ist eine Typklasse. Sie definiert zwei überladene Operatoren == und <=.

```
class Eq a where
    (==), (/=)      :: a -> a -> Bool
    x /= y          =  not (x == y)
    x == y          =  not (x /= y)
```

- Ungleichheit <= ist standardmäßig durch == definiert und umgekehrt.
- Eine Instanziierung muss wenigstens eine der beiden Operationen implementieren:

```
instance Eq Bool where
  True  == True   = True
  False == False  = True
        ==        = False
```

Gleichheit auf eigenen Datentypen

- Gleichheitstests für Datentypen sind schematisch:

```
data Expr = Number Int | Plus Expr Expr

instance Eq Expr where
  Number i    == Number i'     =  i == i'
  Plus e1 e2 == Plus e1' e2' =  e1 == e1' && e2 == e2'
```

- Diese Instanz kann GHC für uns berechnen.

```
data Expr = Number Int | Plus Expr Expr
          deriving Eq
```

- Im Gegensatz zu SML gibt es keine festverdrahtete Druckfunktion für benutzerdefinierte Datentypen. Beispieleingabe: `Number 3`

```
No instance for (Show Expr)
  arising from a use of ‘print’
Possible fix: add an instance declaration for (Show Expr)
```

Typklasse Show zum Darstellen von Werten

- Eine Standarddruckfunktion ist automatisch herleitbar:

```
data Expr = Number Int | Plus Expr Expr
          deriving (Eq, Show)
```

- Meist implementiert man jedoch eine lesbarere Form:

```
instance Show Expr where

  show (Number i)  = show i
  show (Plus e e') = parens $ show e ++ " + " ++ show e'

    where parens s = "(" ++ s ++ ")"
```

Klassen polymorpher Typen

- Naive Instanziierung von Typklassen für polymorphe Typen schlägt fehl.

```
data Maybe a = Just a | Nothing

instance Eq (Maybe a) where
  Just a   == Just a' = a == a'
  Nothing  == Nothing = True
  _        == _       = False
```

```
No instance for (Eq a)
  arising from a use of '=='
In the expression: a == a'
In an equation for '==': (Just a) == (Just a') = a == a'
In the instance declaration for 'Eq (Maybe a)'
```

- In der Tat, eine Funktion
  `(==) :: Maybe a -> Maybe a -> Bool` können wir nicht implementieren.

Typklassen-Annahmen

- Auf einem unbekannten Typen a können wir keine Gleichheit implementieren, sie muss gegeben sein.

```
eqMaybe :: (a -> a -> Bool) -> Maybe a -> Maybe a -> Bool
eqMaybe eqA (Just a)  (Just a')  = eqA a a'
eqMaybe eqA Nothing   Nothing    = True
eqMaybe eqA _         _          = False
```

- Analog können wir **Typklassenannahmen** (engl. *type class constraints*) machen.

```
instance Eq a => Eq (Maybe a) where
  ...
```

Wenn a Instanz von **Eq** ist, dann auch **Maybe** a.

Typklassen benutzen

- Die Funktion **group** `l` fasst aufeinanderfolgende gleiche Elemente der Liste `l` zusammen.

```
group :: Eq a => [a] -> [[a]]
group []                                  = []
group (x:xs) | (ys,zs) <- span (x==) xs = (x:ys) : group zs

group "Mississippi" == ["M","i","ss","i","ss","i","pp","i"]
```

- Diese polymorphe Funktion kann nur für Typen `a` benutzt werden, die die Typklasse **Eq** instanziieren.
- Zur Laufzeit wird für **Eq** `a` ein **Wörterbuch** (engl. *dictionary*) übergeben, das die Implementierungen von der Klassenmethoden == und /= für den Typen `a` enthält.

Simulation von Typklassen in SML

- Eine Typklasse ist ein **Verbundtyp** (engl. *record type*).

```
type 'a eq =
  { eq   : 'a -> 'a -> bool (* equality   *)
  , ineq : 'a -> 'a -> bool (* inequality *)
  }
```

- Instanzen sind *Verbände* (engl. *records*) diese Typs.

```
val intEq : int eq =
  { eq   = fn x => fn y => x = y
  , ineq = fn x => fn y => x <> y
  }

val charEq : char eq =
  { eq   = fn x => fn y => x = y
  , ineq = fn x => fn y => x <> y
  }
```

Simulation von Typklassen in SML

- Eine Typklass-polymorphe Funktion erwartet einen Typklass-Parameter.

```
fun group (aEq : 'a eq) ([] : 'a list) = []
  | group (aEq : 'a eq) (x :: xs)  =
      let val (ys, zs) = span (fn y => #eq aEq x y) xs
      in  (x :: ys) :: group aEq zs
      end
```

- Die Typklass-Instanz wird dann manuell übergeben.

```
val testGroup = map String.implode
     (group charEq (String.explode "Mississippi"))
```

```
val testGroup = ["M","i","ss","i","ss","i","pp","i"] : string list
```

Simulation von Typklassen in SML

- Polymorphe Typklass-Instanzen bilden *records* auf *records* ab.

```
fun eqList (eqA : 'a eq) : 'a list eq =
  let fun eqL [] []                   =  true
        | eqL (x :: xs) (y :: ys) =  #eq eqA x y andalso
                                        eqL xs ys
        | eqL _ _                     =  false
  in { eq   = eqL
     , ineq = fn xs => fn ys => not (eqL xs ys)
     }
  end
```

Zusammenfassung Typklassen

- Überladung wird in Haskell realisiert durch Typklassen und deren Instanzen.
- Instanzen werden an Typklass-polymorphe Funktionen als implizite Parameter (dictionaries) übergeben.
- Welches *dictionary* übergeben werden muss, findet der Übersetzer durch die Typisierung heraus. (Typinferenz erledigt in diesem Fall Arbeit für den Programmierer.)
- Konsequenz: es kann nur eine Instanz pro Typ geben.
- Manchmal möchte man verschiedene Gleichheitsbegriffe wie z.B. Listenidentität oder Gleichheit modulo Permutation. Dies kann man durch einen Wrapper **newtype** realisieren.

```
newtype PermList a = PL [a]

instance Eq a => Eq (PermList a) where
  PL l == PL l' = sort l == sort l'
```